Java on CUDA architecture

Jan Strnad
Faculty of Information Technology,
Czech Technical University in Prague,
Thákurova 9, 160 00 Prague 6,
The Czech Republic.
strnaj11@fit.cvut.cz

Zdeněk Konfršt
Faculty of Information Technology,
Czech Technical University in Prague,
Thákurova 9, 160 00 Prague 6,
The Czech Republic.
konfrst@fit.cvut.cz

ABSTRACT
This paper presents technologies, programs and Java libraries which allow usage of a CUDA capable GPU device within Java programming language. All these approaches are described with their simplified usage guide. Then, we compare performance of these methods, as well as we compare their friendliness to a programmer, documentation or their maturity. For performance tests, we used matrix multiplication and Gamma correction. We recommend to use JCuda library as currently the best available method, followed closely by JNI.

KEY WORDS
CUDA, GPGPU, GPU, Java, JCuda

INTRODUCTION

Compute Unified Device Architecture (CUDA) is a general-purpose computing on graphics processing units (GPGPU) framework and a hardware architecture by Nvidia. The goal of GPGPU paradigm is to use a graphics processing unit (GPU) as a code execution device. This approach allows us to exploit massively parallel nature of the GPU. The design of GPU completely differs from CPU. GPU has its own memory, which is much faster than the main memory. However, the most important difference is in a GPU core architecture. CUDA GPU cores are generally simpler than CPU cores and they typically run at lower frequencies (around 1 GHz). This simple architecture allows embedding thousands cores per one GPU device.

As mentioned above, CUDA is a technology that allows us to benefit from present GPU architecture. For a detailed description of CUDA, some definitions and brief introduction to the CUDA GPGPU programming please refer to [1].

However, this technology is closely tied with programming language called *C for CUDA*. On the other side, Java is one of the most widely used programming language. In this paper we will present possibilities how to utilize CUDA enabled devices using either standard Java tools or alternatively some third party tools and libraries.

BACKGROUND

In this section we will introduce tools that enable Java to cooperate with CUDA. First tool, which is a part of standard Java distribution, is *Java Native Interface (JNI)*. This tool is "a standard programming interface for writing Java native methods and embedding the Java virtual machine into native applications"[2]. This essentially means that we can implement any Java class or method using C(++) languages. With some tweaks that will follow, C for CUDA may be used as well.

OpenMP/Java [3] and *java-gpu* [4] are both tools that are able to generate a code, that utilizes CUDA device, directly from a Java (byte)code.

JCuda [5], *jCUDA* [6] and *Jacuzzi* [7] are standalone Java libraries. These libraries were designed so that they allow indirect communication of Java code and a GPU. By indirect, we mean that these are only responsible for 'routines' (device initialization, memory transfers between CPU and GPU) but the actual computation is defined elsewhere. In fact, these libraries call a CUDA *kernel* which has to be written in C for CUDA.

AVAILABLE METHODS

In this section we provide relatively detailed description of different approaches that use CUDA device from Java language.

As written above, the **JNI** is probably the most straightforward approach: this technology allows one to implement any Java class method using the C or C++ languages. As a first step we need to define a method to implement in C for CUDA using `native` keyword, for example like this:

```
public native void useCUDA();
```

After that, we are able to generate C header files using `javah` command. The last step is an implementation of functions declared by those header files. These may be implemented using C for CUDA language and thus can utilize a CUDA device. An example of vector addition follows:

```
JNIEXPORT void JNICALL add
```

 ISBN 978-80-86943-76-3

```
  (JNIEnv * env, jobject a,
  jobject b, jobject c)
{
  jclass cls;
  long lptr;//for pointer extraction
  int size;//size of vector
  float * arrayA, *arrayB, *arrayC;
  cs = env->GetObjectClass(a);
  // get required attributes
  get_long_field(env, cs, a, "ptr",
                 &lptr);
  arrayA = (float*)lptr;
  // the same for arrays b and c
  get_int_field(env, cls, a, "size",
                &size);
  vector_add<<<grid, block>>>(
    arrayA, arrayB, arrayC, size);
}
```

When compiling to a shared library, one must make sure that generated code is position independent. This is achieved using following command:

```
$ nvcc -shared -Xcompiler -fPIC -o
    libout.so <src.files>
```

In order to start our program, we must declare where Java should look for our shared native library. This is done using `-Djava.library.path` argument. Detailed description of these steps can be found in [8], page 42.

Second category of available methods uses completely different approach. Unlike the JNI, which is very low-level, this category does not require any knowledge about CUDA and CUDA programming at all. This means that methods from this category take a regular Java code as an input and transform it to a form that uses a CUDA device.

However, these tools do not work fully automatically. The piece of code which should be parallelized must be marked in some way. There are currently two implementations: OpenMP/Java and java-gpu.

Automatic code generation brings some limitations as well. For example, it is not possible to control utilization of fast shared memory. It is also not possible to exploit features such as warp voting etc.

OpenMP/Java is an implementation of Open Multi-Processing (OpenMP) for Java language, which supports not only CPU as a backend, but a CUDA capable GPU as well. This tool is designed as an extension to the standard Java compiler so it can recognize OpenMP commands. These commands are prefixed with symbol `//` (one line comment). That has one positive side effect – the code can be compiled using standard compiler and functionality of that code remains the same. For example, a simple vector addition can be parallelized using single `for` loop:

```
//#omp parallel for
    shared(a,b,result)
for (int i = 0; i < size; i++)
{
  result[i] = a[i] + b[i];
}
```

, where `a`, `b` and `result` are `float` arrays. An OpenMP/Java source can be compile using `jampc` command. It can be run using `jcuda_java` command. Please refer to project's homepage [3] for details.

On the downside, OpenMP/Java contains some bugs. This is mainly due its experimental focus. For instance, we had troubles with compilation and we had to manually alter its source code. Nevertheless, our patch was quite simple – it just converts `int` to `size_t` in one of its functions. Secondly, OpenMP/Java compiler gives us incorrect warnings. Details including installation instruction are available in [8] as well as in [3].

Similar solution called **java-gpu** uses Java annotations to identify a code to be parallelized. However, we were not able to make it work. Specifically, no CUDA code was generated by this tool and CPU was used as the backend instead.

Last tools to mention are Java **libraries** which wrap the Driver API provided by CUDA. This approach is a kind of opposite to the previous one – it requires a very detailed knowledge of CUDA programming. Basically, a source code utilizing a CUDA device is consisted of two parts. First part is written in C for CUDA language and contains only a code which is executed by the GPU device. Second part is written in Java language and it is responsible for a 'glue' code. This for instance include device initialization, memory (de)allocation, memory transfers and naturally CUDA kernel invocations.

Generally, these steps must be done: *a*) Select a device to work with and initialize it. *b*) Create a new CUDA context. *c*) Load a CUDA module – file with the CUDA code to execute(it can be compiled or in PTX format). *d*) Obtain a CUDA function – the CUDA kernel function, which will be used. *e*) Copy data from the main memory to the GPU memory. *f*) Prepare CUDA kernel's arguments. *g*) Invoke the CUDA kernel and wait for a result. *h*) Copy the result back from the GPU memory to the main memory. More details can be found again in [8].

There is currently one up-to-date implementation of the Driver API wrapper: **JCuda**. Other libraries exist, such as **jCUDA** and **Jacuzzi**. However, these are quite outdated and do not support all features provided by newer devices (e.g. surface memory).

EXPERIMENTS

In order to compare performance of different approaches, we designed performance tests. In our

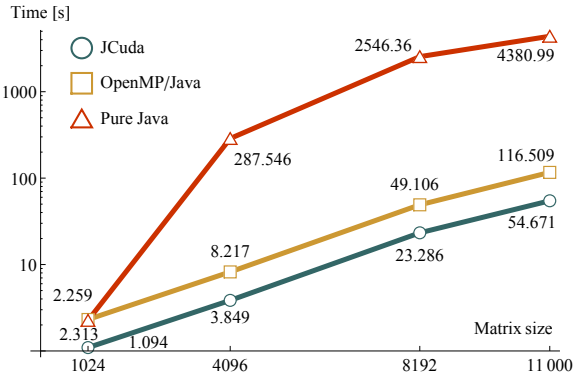

Figure 1: Matrix multiplication performance of different approaches. Note that JNI is covered by JCuda. JNI and JCuda were much faster than OpenMP/Java.

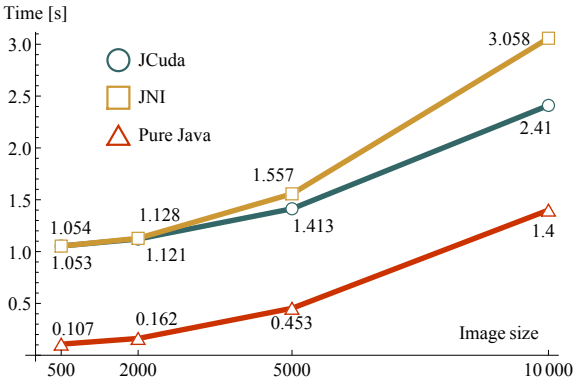

Figure 2: Gamma correction performance of different approaches. OpenMP/Java was not included because it does not support `pow` function.

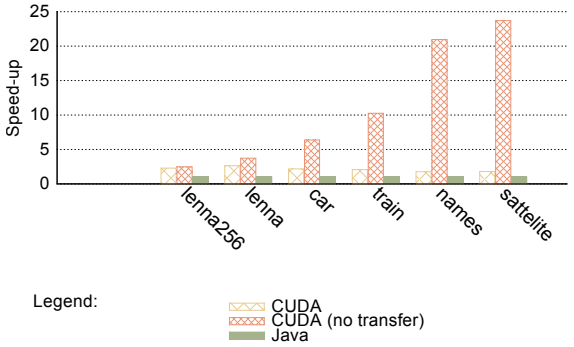

Figure 3: The impact of memory transfers on overall speed-up in Gamma correction algorithm. Memory transfers consumes most of the time.

experiments we used PC with following configuration: Intel Core i5 760 (2.8GHz, 4 cores, 8MB cache), 4GB RAM, Nvidia GeForce GTX 480, Ubuntu 11.04 and CUDA tookit 4.0.

We chose algorithms from linear algebra and image processing for our performance tests. Specifically, we used matrix multiplication and Gamma correction. We used various input data sizes. For linear algebra, we used matrices with width and height from 1024 to 11000, while all matrices were square shaped. Similarly for images which were also square shaped and had sizes from 500 to 10000 pixels. All test were run three times and average value of those was taken into account.

All algorithms which were implemented using JNI and JCuda were optimized in terms of reducing global memory access using shared memory as described in [1]. This was not possible to achieve using OpenMP/Java. The plot also contains pure Java algorithm (single thread) which does not use a GPU at all.

Figure 1 shows performance of different methods for matrix multiplication algorithm. Results of other algorithms from linear algebra can be found in [8].

The plot does not show performance of JNI. The reason is that it would be covered by JCuda otherwise. Those are the two fastest while OpenMP/Java was the slowest. Still, OpenMP/Java was much faster than pure Java.

Results for Gamma correction are in Figure 2. We excluded OpenMP/Java from this test because it lacked of support for `pow` (power) function which was necessary. This plot shows that JCuda was the fastest for Gamma correction. However, all CUDA based algorithm were slower than pure Java. This is not a fault, because programs that use CUDA also contain a device initialization, memory transfers etc. which are included in a result time. Figure 3 shows an impact of memory transfers on overall speed-up in Gamma correction algorithm.

Aside the results of our performance tests, a development time and effort were noted during our experiments. Our experience and some empirical properties (such as operating system support) resulted in an overall subjective method evaluation. To be more specific, every attribute was scored with numbers $1, 2$ and 3 where 1 is the best. All attributes have the same weight. The best approach was determined as the method with minimal sum of scored attributes. See Table 1. Details and table with operating system support can be found in [8].

Attribute	jCUDA	JCuda	OpneMP/Java	JNI
dev. time	2	2	1	3
friendliness	2	1	3	1
docs	2	1	3	1
status	3	1	3	1
Total points	8	5	10	6

Table 1: Comparison of different approaches. JCuda and JNI provide the best support to a programmer while OpenMP/Java the worst.

 ISBN 978-80-86943-76-3

DISCUSSION

All these approaches can be used to improve performance of Java applications. Nevertheless, there were significant performance differences between different methods. To be more specific, tools that transform Java code automatically such as OpenMP/Java could not utilize advanced CUDA specific features e.g. shared memory.

On the other hand, methods like JNI and JCuda were able to utilize all possible features. As mentioned before, performance was not the only one criterion. Other criteria included productivity, stability and documentation. JNI's productivity was very low – this is mostly because even simple operation like copying value of class variable has to be done in about three steps. We have spent at least twice as much programming time when using JNI than with any other method. Since JNI was a part of a standard Java distribution, it does not include any obvious bugs and was documented very well.

OpenMP/Java was a complete opposite of JNI. It had very high productivity (e.g. `for` cycle could be parallelized using one line of code) but it contained some bugs such as incorrect warnings. There was no documentation available. Another evidence of impractical usability is a lack of support for basic mathematic functions such as `pow`. Note that only Linux was supported.

JCuda provides a level of productivity which was generally greater than JNI's but exceptions exist. For example, a device initialization has to be handled manually and also CUDA kernel invocation was more complicated (kernel arguments have to be copied manually). This was caused by a requirement to use the Driver API. JCuda is being developed very actively and it is documented quite well.

CONCLUSION

This paper listed and briefly described solutions for cooperation between CUDA GPGPU technology and Java programming language. These solutions were divided into three categories. First category was JNI. Second category was based on an automatic CUDA code generation from a Java (byte)code. Last solution utilized a Java library which wrapped the standard CUDA Driver API.

JNI and Java libraries were not easy to develop with but they had very good performance. On the other hand, second category was fairly easy to develop with but final performance was not as good as the former. Second category was not production ready yet.

We recommend to use JCuda library. The reason for it is that less development time is required when using JCuda in comparison to JNI. This library is documented quite well, supports various types of operating systems, is production ready and overall performance is identical with JNI. There is currently no other solution which can be both time efficient (in terms of development time) and powerful.

In the matter of future work one can explore another solutions. The first on is Rootbeer. Rootbeer provides additional abstraction so that it can run complex Java objects on a GPU [9].

There are also tools that target more general OpenCL architecture [10] instead of CUDA. There are at least two projects worth trying: JavaCL [11] and Aparapi [12]. Another alternative is ScalaCL that is designed for another JVM language called Scala. This project tries to develop domain specific language that translates to OpenCL code during compilation time [13].

REFERENCES

[1] *CUDA Programming Guide*, Nvidia. [Online]. Available: http://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html

[2] Java Native Interface. Oracle corp. [Online]. Available: http://docs.oracle.com/javase/7/docs/technotes/guides/jni/index.html

[3] M. Philippsen, R. Veldema, M. Klemm, G. Dotzler, and T. Blaß, *OpenMP/-Java*, Friedrich-Alexander University Erlangen-Nuremberg. [Online]. Available: https://www2.informatik.uni-erlangen.de/EN/research/JavaOpenMP/index.html

[4] P. Calvert, *java-gpu*, Computer Laboratory, University of Cambridge. [Online]. Available: http://code.google.com/p/java-gpu/

[5] jcuda.org, *JCuda library*. [Online]. Available: http://jcuda.org/

[6] *jCUDA*, Hoopoe. [Online]. Available: http://www.hoopoe-cloud.com/Solutions/jCUDA/Default.aspx

[7] A. Heusel, *Jacuzzi*. [Online]. Available: http://sourceforge.net/apps/wordpress/jacuzzi/

[8] J. Strnad, "Java on CUDA architecture," Banchelor's thesis, Czech Technical University in Prague, Faculty of Information Technology, 2012. [Online]. Available: https://dip.felk.cvut.cz/browse/pdfcache/strnaj11_2012bach.pdf

[9] T. Kiefer and A. Miftah. Rootbeer GPU compiler. [Online]. Available: http://rbcompiler.com

[10] *OpenCL*, The Khronos Group. [Online]. Available: http://www.khronos.org/opencl/

[11] JavaCL. JavaCL team. [Online]. Available: http://code.google.com/p/javacl

[12] Aparapi. Aparapi team. [Online]. Available: https://code.google.com/p/aparapi

[13] ScalaCL. ScalaCL team. [Online]. Available: https://github.com/ochafik/ScalaCL